# Parallel Design for Item-Based Movie Recommendation

Ling Feng, Zongjun Liu, Guangya Wan, and Di Zhen

Harvard University

May 5, 2022

**Abstract**

Collaborative filtering recommendation system has been widely used in various network platforms, especially in Movie Recommendation system. Spurred by the rapid development of information technology such as cloud computing and data center, an increasing number of movie and users' data are emerging on the Internet, so it's essential for the collaborative filtering algorithm, which utilize those big data to get a similarity score and recommends the movie to new user, to run fast and scale well in real life applications with sufficient computational resources. In this project, we will build and explore the bottleneck of unparallel item-based collaborative filtering recommendation algorithm and improve it based on the techniques we learned in class. We first obtained raw data on movies and user ratings from public source,built a pipeline which cleans, reads the data and run the collaborative filtering algorithm, and finally used hybrid methods which takes advantage of both OpenMP and MPI to improve our pipeline. The algorithm discussed in this project is different from the other common implementation in this field as we only include data on user ratings and movie ID and need to use cosine similarity to compute the similarity score. This project also incorporates roofline analysis and strong scaling analysis that justifies the usage of our parallel design and how it can provide improvement on practical recommendations. In sum, our parallel design have shown significant boost on speedup and efficiency, especially for when calculating similarity of movies and merge sort. Although we failed to improve our data input compared to serial code, we demonstrated that as data size gets larger, eventually the boost in speed our parallel design would out-weight the cost created by communication overhead. We hope our simple and effective parallel approach will serve as a solid baseline and help ease future research in speeding up the collaborative filtering recommendation system.

## 1 Background and Significance

Big data has emerged as a widely recognized trend, attracting attentions from government, industry and academia.Generally speaking, Big Data concerns large-volume, complex, growing data sets with multiple, autonomous sources. Big Data applications where data collection has grown tremendously and is beyond the ability of commonly used software tools to capture, manage, and process within a "tolerable elapsed time" is on the rise. The most common challenge for the Big Data applications is to explore the large volumes of data and extract useful information or knowledge for future actions.

One of the most commonly used application is movie recommendation system, which assists users in a decision making process where they want to choose some movies among a potentially overwhelming set of alternative options. Collaborative filtering such as item- and user-based methods are the dominant techniques applied in recommendation system. The basic assumption of user-based Collaborative filtering

is that people who agree in the past tend to agree again in the future. Different with user-based Collaborative filtering, the item-based Collaborative filtering algorithm, which will be the focus of our project here, recommends a user the items that are similar to what he/she has preferred before.

The reason why we picked item-based collaborative filtering is that it has several advantage compared to other methods like user-based collaborative filtering or matrix factorization approach [9]. 1: The performance of item-based recommendation algorithm keeps improving as data sizes are increased, so it's very suitable for big-data application. 2: Context independent(so it's easy to migrate from one system to another) and easy to implement (which suites our project). 3: Very robust and tends to give good accuracy.

Although traditional Collaborative filtering techniques are sound and have been successfully applied in many real life applications,they encounter two main challenges for big data application: 1) to make recommendation decision within acceptable time; and 2) to generate ideal recommendations from so many movies and users. Concretely, as a critical step in item-based Collaborative filtering algorithms, to compute similarity between every pair of users or movies may take too much time as the data size gets larger. As more and more data on movies and user are being collected in the infrastructures,companies must now face the unprecedented difficulties to avoid lose its timeliness when finding ideal movie recommendations fast enough to provide users smooth experiences.

Thanks to the the recent development on computer architecture and lower price on more powerful chips, using parallel and distributed computing techniques can easily solve such problems at a reasonable cost. We will show here by designing parallel programs, we can still save a lot of time and money by sorting through "big data" faster than ever.

## 2 Scientific Goals and Objectives

The purpose of our project is to build an item-based collaborative filtering movie recommendation model with shared memory support OpenMP and distributed memory support MPI to improve the efficiency of movie recommendation process.

Specifically, we aim to implement item-based collaborative filtering algorithm with distributed-memory and shared-memory parallelism, train and evaluate the model on MovieLens 1M dataset[2]. This dataset contains 1,000,000 ratings from 6,000 users on 4,000 movies. It was collected by GroupLens Research over various periods of time from MovieLens website, which is based on a well-known recommender system using collaborative filtering of movie ratings and review, as well as a virtual community. By speeding up data input and processing, similarity computation, and merge sort algorithm, we aim to demonstrate significant improvement of overall efficiency of our movie recommendation model.

## 3 Algorithms and Code Parallelization

The memory-based technique of item-based collaborative filtering for movie recommendations is computing similarities between pair of movies[3]. There are a lot of similarity metrics, such as pearson, cosine, Euclidean, etc. Pearson similarity is able to handle data that's subject to different ratings scales. Euclidean is good at dealing with data that's not sparse. Our choice is to use cosine similarity metric to measure similarities, because it is capable of handling sparse data with many ratings undefined [4]. The cosine similarity formula is as follows, for movie $i$ and movie $j$ among all $U$ users. By iterating through

all pairs of items, we have a matrix of values that indicates how close other movies are to each movie.

$$cosine\_similarity(i,j) = \frac{\sum_u^U r_{(r,i)} r_{(u,j)}}{\sqrt{\sum_u^U r_{(u,i)}^2}\sqrt{\sum_u^U r_{(u,j)}^2}}$$

where $U$ is the number of users, $u$ is a user for which we are generating recommendation, $r_{(u,i)}$ is the rating of user $u$ for movie $i$.

To be able to recommend movies to users, we use the similarity matrix and users' past movie ratings to calculate recommendation scores for a target item and an active user. This can be easily achieved by the following equation:

$$score(u,i) = \frac{\sum_j^I \left[cosine\_similarity(i,j) \cdot r_{(u,j)}\right]}{\sum_j^I cosine\_similarity(i,j)}$$

where the numerator is the sum of the multiplication of movie $i$ and movie $j$'s similarity and rating by user $u$ to movie $j$, the denominator is the sum of similarity of movie $i$ and movie $j$.

To make efficient movie recommendations with scores, we also apply merge sort to sort the scores of movies so as to give optimal recommendations to users, from most preferable to less preferable. Merge sort is a divide-and-conquer algorithm with time complexity of $O(n \log n)$. It works as follows: first, divide the unsorted list into 2 parts; second, call mergesort recursively on each of the parts and then merge each of the sorted parts. We chose merge sort algorithm because it's naturally balanced and symmetric, and promising to be paralleled by task level parallelism. Alternative sorting algorithms are also good candidates for this sorting task, such as bubble sort and quick sort.

With these ideas, as the main developers, we developed a movie recommendation model with three main parts in C++ code: reading in the MovieLens dataset including movies and ratings, computing movie similarity matrix, and merge sorting the scores. After testing our serial code and observing the runtime for each part, we identified the bottleneck is computing similarity matrix which takes a lot of time (181 seconds for serial baseline). We also found great opportunities on improving the efficiency of reading in data, computing similarity matrix, and merge sort, so we parallelized the code by using OpenMP and MPI.

### Parallelism 1: Data Input

In this project, we present a hybrid MPI and OpenMP parallelization scheme for item-based movie recommendation. First, we applied MPI to parallelize the process of reading and storing dataset for training the recommendation model. Because one concern of poor performance of our application is that reading in and storing large dataset that is of size 1,000,000 is computationally expensive. Considering that MPI is a good setting for parallel I/O, we expect less time to consume in the data loading part. Specifically, we allow each process to take care of a subset of data. After finishing reading in all data in different processes, we combined those subsets from all processes and distributed the them back to all processes by MPI AllReduce routine.

### Parallelism 2: Similarity Matrix

Second, we applied MPI and OpenMP to compute similarity in parallel and applied MPI reduction to combine local similarities to final similarity matrix at the end. As we observed in the serial code testing

that computing similarity matrix is the most time-consuming part of our model, we expect to see great improvement of performance with the support of MPI. Specifically, each processor will compute a part of the similarity matrix, and these intermediate results will be combined to produce the whole similarity matrix by calling MPI Reduce.

### Parallelism 3: Merge Sort

Third, we applied OpenMP tasking to parallelize merge sort, since task parallelism is well suited to the expression of nested parallelism in recursive divide-and-conquer algorithms and of unstructured parallelism in irregular computations. Tasks are queued and executed whenever possible at the task scheduling points. Synchronization between tasks is achieved using the taskwait pragma. Another way to parallelize merge sort is to use sections construct. Rather than maintaining a task queue, sections construct organizes all tasks into their own groups.

### Validation, Verification

To verify the parallelization performance, we compare the wall time of serial implementation and parallel implementation with different numbers of threads and processors. A negative relationship between wall time and the number of processors indicates that by using MPI or OpenMP we successfully speed up the model. In addition, we measure the wall time of each target part of the code (i.e. data input, similarity computation, and merge sorting) before and after parallelization to confirm the speed-up of each part. The result of our experiments are reproducible on a Broadwell node. Instructions can be found on our Github repository.

There are several ways to validate our model. One way is to switch dataset. By using MovieLens 25 million dataset, rather than 1 million dataset, we expect to see significant improvement of run time. Using a new dataset with different source such as Netflix dataset [5] used for the Netflix Prize competition is also a good validation approach. Second, we can compare our model with other implementations of parallel recommender systems as well as packages [6, 7, 8].

## 4 Performance Benchmarks and Scaling Analysis

### 1. Data Input

#### (a) Baseline and Efficiency Analysis

We first try to run MPI parallel file read algorithm on a single processor (thus can be treated the same as running a serial code), and the performance of the result will be used as our baseline. Next, we rerun the algorithm instead using 10 processors. We first divided the data into 10 sub-files, and each of the processor will read those sub-file, and finally merge all of them together by using MPI's AllReduce routine. We calculated run time, speedup, and efficiency for both methods, and summarize them in the figure and table below.

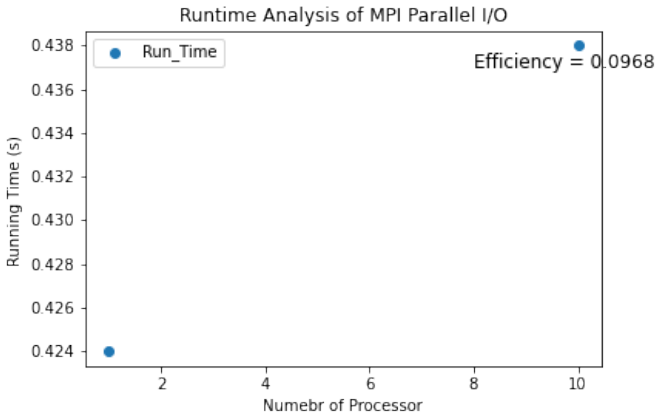


Figure 1: Run Time analysis of File Input.

| N | Time | Speed up | Efficiency |
|---|---|---|---|
| 1 | 0.424 | 1.000 | 1.000 |
| 10 | 0.438 | 0.966 | 0.097 |

Table 1: Scaling Analysis Result of Reading Data Input.

As we can see from the table above and (Figure 1) , increase the number of processors here even increases the run-time in our algorithm.

### (b) Explanation

We can see from the above result that we did not actually make an improvements by using more processors in our MPI parallel design of reading and storing files compared to using a single processor. The main reason is that the data size of our current sample is small (only 24.6 Megabytes), and the speedup boosted by adding more processors does not outweigh the its accompanying overhead associated with the amount of time threads spend communicating with each other, in such a small data-set. As the data size gets larger, say more than several Gigabytes, each process should be able to take more work and leads to a large reduction in data input time. It is expected that our parallel algorithms should find a sweet spot compared to the serial code with a larger dataset.

### (c) Future Work

Since our dataset is comparatively small, in the near future, we would like to replicate current user ratings and intentionally enlarge the data file, to test the performance of our parallelized data input application.

## 2. Similarity Matrix

### (a) Roofline Analysis

We first performed roofline analysis on our model to show the performance of serial and parallel computation of similarity matrix. The walltime of our serial code of this part on a single thread Broadwell node is 181 seconds. With the support of OpenMP, it's able to run on 4 threads and achieves a walltime of 66.67 seconds. Figure 2 shows that the computation of similarity matrix is within memory bound, and

thread-level parallel implementation achieves better performance.

For a Broadwell node with 32 cores, it is possible to further exploit the thread level parallelism by increasing the number of threads thus further shift the position of this kernel horizontally in the roofline analysis graph, but increasing the number of threads may lead to thread imbalance and over-subscription.

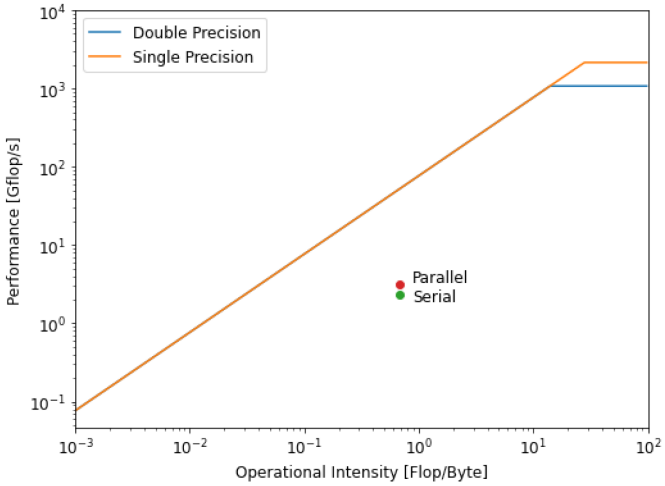


Figure 2: Roofline Analysis of Similarity Matrix.

### (b) Strong Scaling Analysis

For the strong scaling analysis in the following part, we choose to set the number of threads to 4 to avoid over-subscription when number of processors increases. The serial code paralleled with 4 threads is used as the benchmark for calculating the speedup.

We performed strong scaling analysis on parallel computation of similarity matrix using MPI. The walltime of our parallel code of this part on 4 threads achieves a walltime of 66.670 seconds on a single processor. The computation is able to run on N processors and each with 4 threads. The walltime are recorded, and speedup and efficiency are calculated by using Amdahl's Law (Table 2). We visualize the relationship between strong speedup and the number of processors, compared to the ideal speedup line (Figure 3). We observe a increasing speedup with the increasing number of processors below 8 processors. When using 10 processors, there is a drop in both speedup and efficiency.

| N | Time | Speed up | Efficiency |
|---|---|---|---|
| 1 | 66.670 | 1.000 | 1.000 |
| 2 | 28.202 | 2.364 | 1.182 |
| 4 | 19.193 | 3.474 | 0.868 |
| 8 | 10.076 | 6.617 | 0.827 |
| 10 | 11.168 | 5.970 | 0.597 |

Table 2: Scaling Analysis Result of Computing Similarity Matrix.

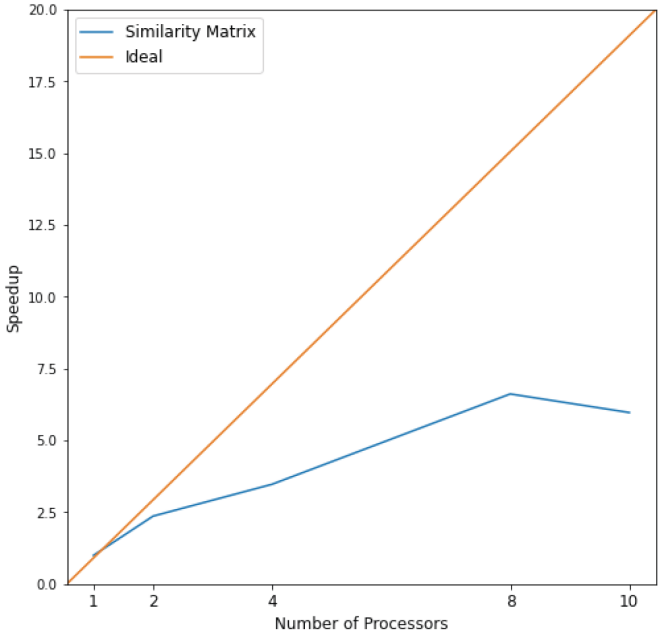

Figure 3: Speedup of Computing Similarity Matrix.

This drop in speedup when $n\_processor = 10$ could be attributed to the overhead of communicating large matrix between processors. The size of similarity matrix is 4000x4000, and its size increases quadratically as the number of movies increases. This overhead also leads to a less than ideal speedup.

## 3. Merge Sort

### (a) Cross-over Point and Performance

To better improve the performance of mergesort, we performed experimental testing and found that the cross over point for mergesort is when the number of elements in the vector is over 10000. In other words, if the number of elements in the vector is less than 10000, serial mergesort is faster. Otherwise, parallel mergesort prevails. The wall time for serial mergesort is consistently less than the wall time for parallel mergesort with OpenMP, which ranges from 0.007 to 0.008 seconds. Overall, the performance is not unexpected because we only have 4000 datapoints which are not enough for parallel mergesort to demonstarte its efficiency. Moreover, it is also likely the case that any benefits we gained from parallelizing the code might be very well offset by the higher overhead of task queuing and scheduling.

### (b) Future work of Mergesort with MPI

With OpenMP, the tree of recursive calls is automatically mapped to threads because threads are assigned dynamically to parallel regions. However, in MPI, all processes start at the beginning of the program and run the same code simultaneously. Therefore, we must tweak the MPI program a bit so that it allows each process to map to its correct node in the recursion tree. In fact, the above differences make the task of mergesort with MPI significantly harder than with OpenMP.
When MPI maps processes to nodes in the recursion tree, it will construct a pseudo process tree, where process $P_0$ is the root of the tree and the rest of processes are the nodes of the tree. The root process divides the data in half and forwards half of it to an auxiliary process which returns the sorted data to the root process. The other half of data is kept by the root process for future sorting using the same

procedure. After the data is sorted, the root process will merge two halves of data. The illustration of the procedure is below:

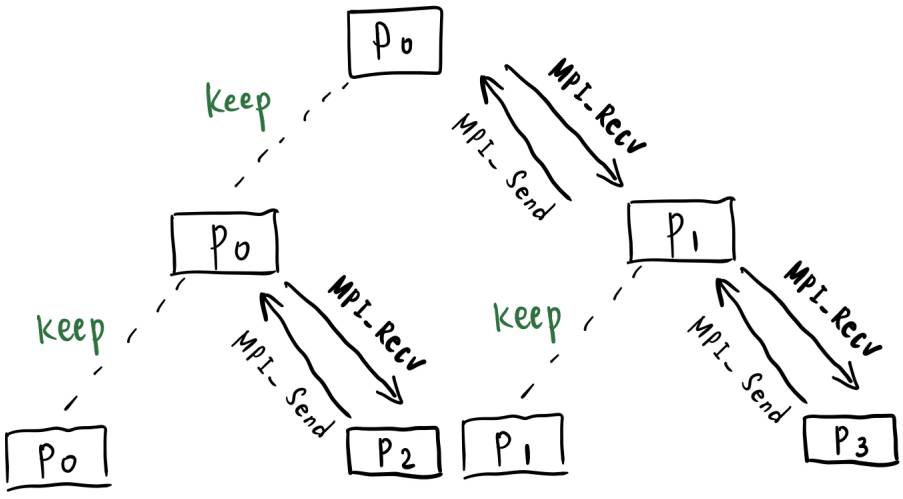


Figure 4: MPI demo

## 5 Resource Justification

After being optimized by shared-memory parallelism (OpenMP) and distributed-memory parallelism (MPI), computing the movie recommendation for a particular user only takes around 10 seconds with one node on Harvard computing cluster. Considering that the computed similarity matrix can be used to produce movie recommendation for other users within only several milliseconds, we are not requesting computing resources on Harvard computing cluster now.

For future work, we would like to collect larger dataset, say user rating records up to gigabytes, to further investigate the performance of MPI parallelized file input and OpenMP parallelized merge sort. We may request computing hours at that time.

## References

[1] Leslie Lamport. *LaTeX: a document preparation system.* Addison-Wesley, Reading, Massachusetts, 1993.

[2] MovieLens dataset, http://grouplens.org/datasets/movielens/

[3] Xiaoyuan Su and Taghi M. Khoshgoftaar. *A survey of collaborative filtering techniques.* Advances in Artificial Intelligence. Vol. 2009, January 2009.

[4] Emmanouil Vozalis and Konstantinos Margaritis. *Analysis of recommender systems' algorithms.* The 6th Hellenic European Conference on Computer Mathematics its Applications (HERCMA). Athens, Greece, pages 732–745, 2003.

[5] Netflix Prize. http://www.netflixprize.com/.

[6] Wang, Jun and Pouwelse, Johan and Lagendijk, Reginald L. and Reinders, Marcel J. T. *Distributed Collaborative Filtering for Peer-to-Peer File Sharing Systems*. Association for Computing Machinery. Dijon, France, pages 1026–1030, 2006.

[7] Miller, Bradley N. and Konstan, Joseph A. and Riedl, John. *PocketLens: Toward a Personal Recommender System*. Association for Computing Machinery. New York, NY, USA, Vol. 22, 3, pages 437–476, 2004.

[8] Berkovsky, Shlomo and Kuflik, Tsvi and Ricci, Francesco. *Distributed Collaborative Filtering with Domain Specialization*. Association for Computing Machinery. Minneapolis, MN, USA. Pages 33-40, 2007.

[9] Byström, H. *Movie Recommendations from User Ratings*, Stanford University (2013).